Introduction to Junit 4 and Junit 5 and Mockito

Introduction to Junit 4

Lesson Objectives

- After completing this lesson, participants will be able to
  - Understand importance of unit testing
  - Install and use JUnit 4
  - Use JUnit Within Eclipse

Why is Testing Necessary

- To test a program implies adding value to it.
  - Testing means raising the reliability and quality of the program.
  - One should not test to show that the program works rather that it does not work.
  - Therefore testing is done with the intent of finding errors.
- Testing is a costly activity.

What is Unit Testing

- The process of testing the individual subprograms, subroutines, or procedures to compare the function of the module to its specifications is called Unit Testing.
  - Unit Testing is relatively inexpensive and an easy way to produce better code.
  - Unit testing is done with the intent that a piece of code does what it is supposed to do.

What is Test-Driven Development (TDD)

- Test-Driven Development, also called Test-First Development, is a technique in which you write unit tests before writing the application functionality.
  - Tests are **non-production code** written in the same language as the application.
  - Tests return a simple **pass** or **fail**, giving the developer immediate feedback.

Why Unit Testing

- You can cite following reasons for doing a Unit Test:
  - Unit testing helps developers find errors in code.
  - It helps you write better code.
  - Unit testing saves time later in the production/development cycle.
  - Unit testing provides immediate feedback on the code.

What is JUnit

- JUnit is a free, open source, software testing framework for Java.
- It is a library put in a jar file.
- It is not an automated testing tool.
- JUnit tests are Java classes that contain one or more unit test methods.

Why JUnit

- JUnit allows you to write tests faster while increasing quality and stability.
- It is simple, elegant, and inexpensive.
- The tests check their own result and provide feedback immediately.
- JUnit tests can be put together in a hierarchy of test suites.
- The tests are written in Java.
- JUnit allows you to write tests faster while increasing quality and stability.
- It is simple, elegant, and inexpensive.
- The tests check their own result and provide feedback immediately.
- JUnit tests can be put together in a hierarchy of test suites.
- The tests are written in Java.

Steps for Installing JUnit

- Following are the steps for installing and running JUnit:
  - Download JUnit from www.junit/org. You can download either the jar file or the zip file.
    - Unzip the JUnit zip file
  - Add the jar file to the CLASSPATH.
    - Set CLASSPATH=.,%CLASSPATH%;junit-4.3.1.jar

Using JUnit within Eclipse

- JUnit can be easily plugged in with Eclipse.
- Let us understand how JUnit can be used within Eclipse.
  - Consider a simple “Hello World” program.
  - The code is tested using JUnit and Eclipse IDE.
- Steps for using JUnit within JUnit:
  - Open a new Java project.
  - Add junit.jar in the project Build Path.

Using JUnit within Eclipse (Contd.)

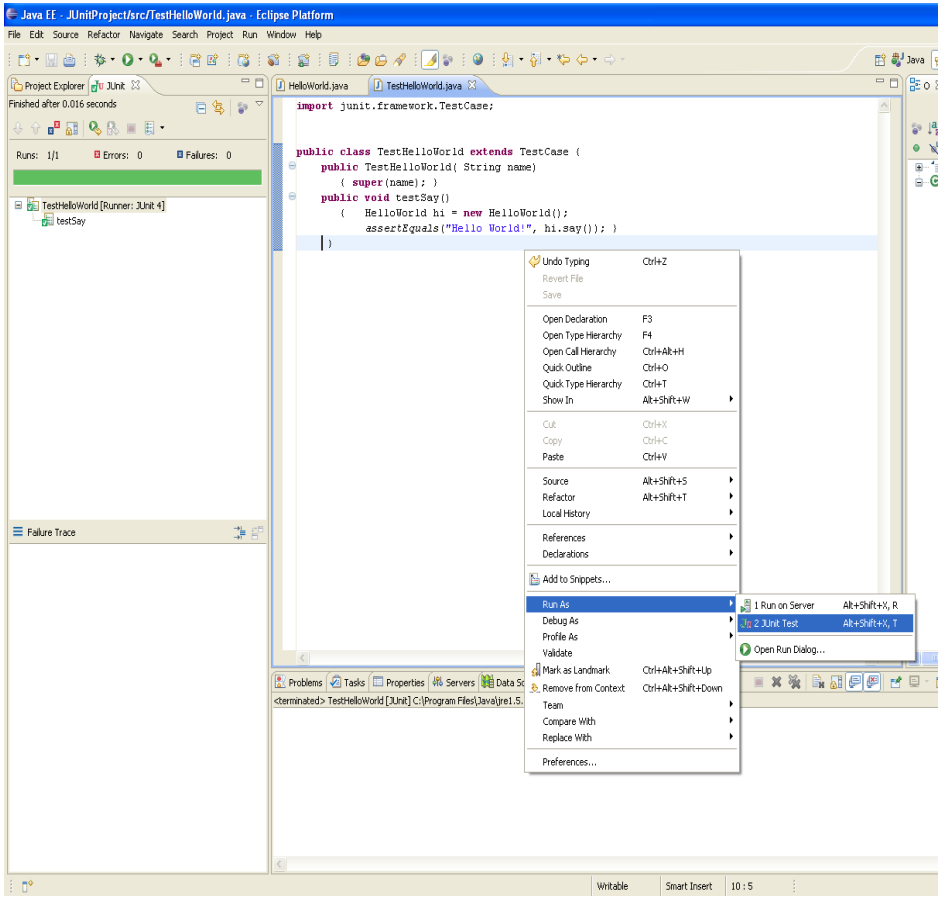
- Write the Test Case as follows:

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestHelloWorld {
    @Test
     public void testSay()        {
          HelloWorld hi = new HelloWorld();
          assertEquals("Hello World!", hi.say());
       } }
```

```
class  HelloWorld{
     String say(){
          return “Hello World!”}
}
```

Using JUnit within Eclipse (Contd.)

- Run the Test Case.
  - Right-click the Project → Run As → JUnit Test
- The output of the test case is seen in Eclipse.



Annotation Types in JUnit4.x

- JUnit4.x introduces support for the following annotations:
  - @Test – used to signify a method is a test method
  - @Before – can do initialization task before each test run
  - @After – cleanup task after each test is executed
  - @BeforeClass – execute task before start of tests
  - @AfterClass – execute cleanup task after all tests have completed
  - @Ignore – to ignore the test method

Example Using JUnit

- Create a new Java project called *junit-app*
- *Create package, com.sapient.service in src folder, add create the following class in it.*

```
public class HelloWorld {
private String message;

public HelloWorld() {
this.message="Hello World!!";
}

public HelloWorld(String message) {
this.message=message;
}

public String getMessage() {
return message;
}

public void setMessage(String
message) {
this.message = message;
}

public String myMessage() {
return  "Hi!"+this.message;
}
}
```

Example Using JUnit

•Create a new source folder ***test*** **(right-click on the project and select *New → Source Folder*)**

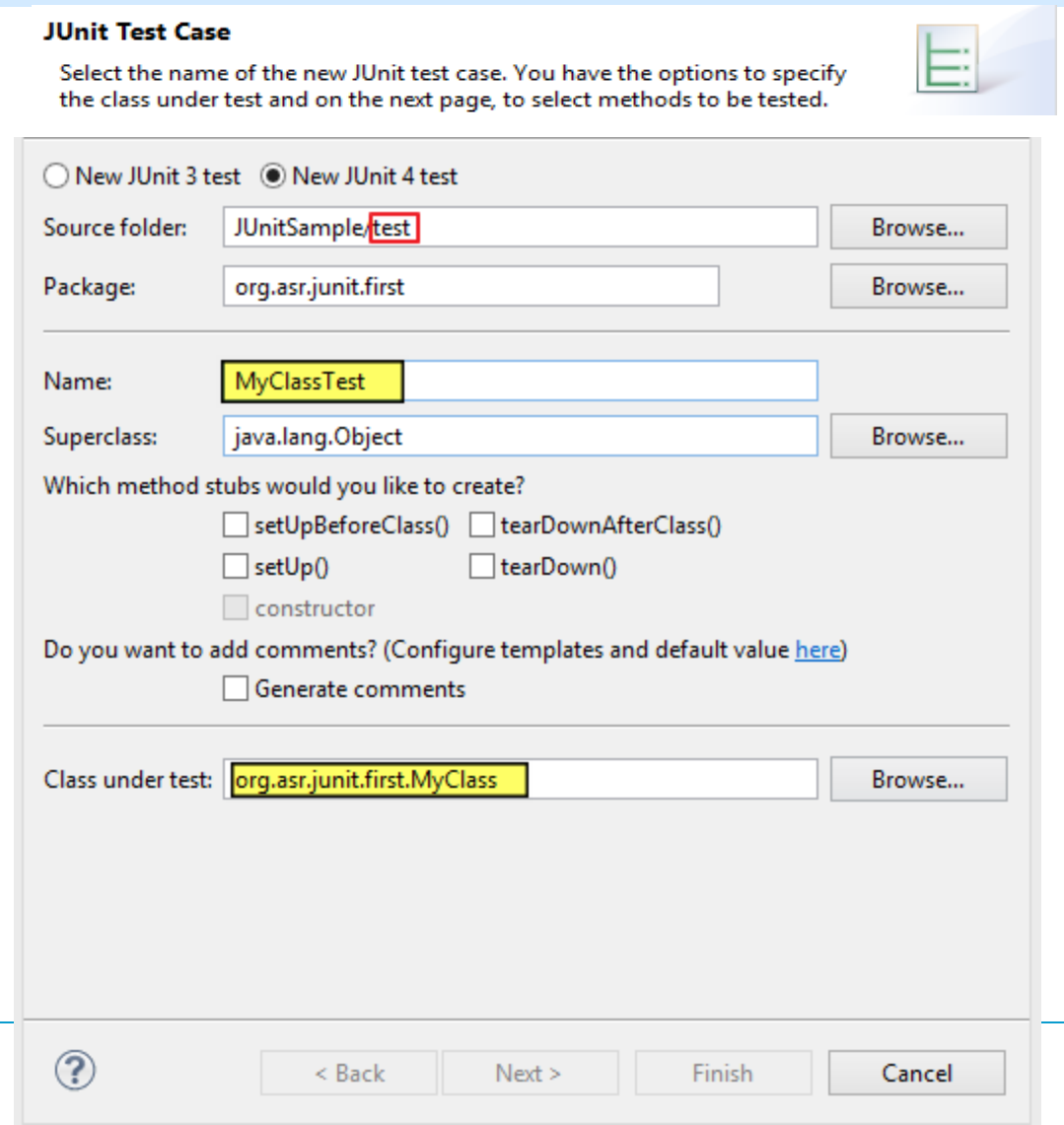
•Create *com.sapient.service* ***package in*** test folder

Create a JUnit test

Right-click on your class, **`HelloWorld`** in the *Package Explorer* view and select *New → JUnit Test Case*.

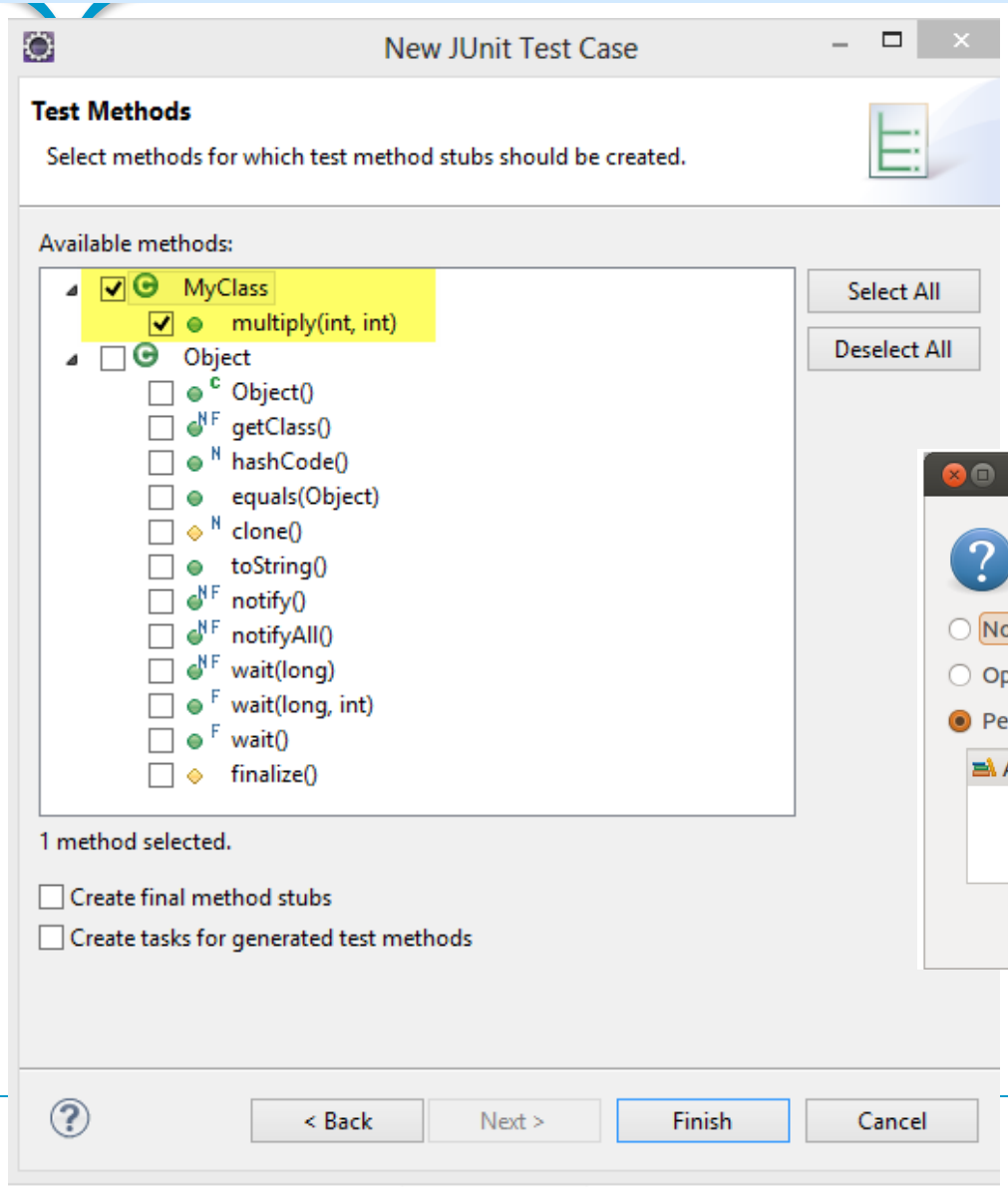
Example Using JUnit

In the following window ensure that the *New JUnit 4 test* flag is selected and set the source folder to ***test***, so that your test class gets created in this folder.

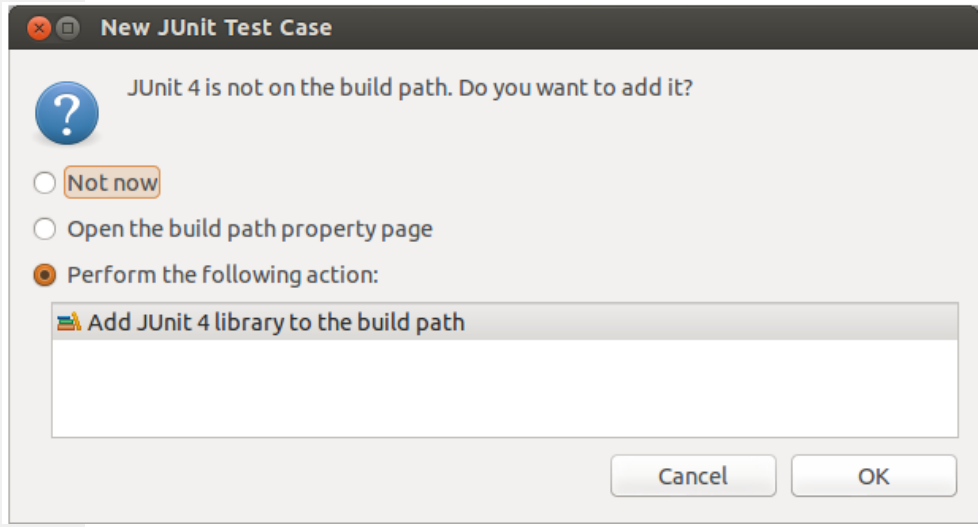


Example Using JUnit

Click on *Next* button and select the methods that you want to test.



If the JUnit library is not part of the classpath of your project, Eclipse will prompt you to add it.
Use this to add JUnit to your project.



HelloWorldTest

```
public class HelloWorldTest {

@Before
public void beforeMethod() {
System.out.println("Before method");
}

@Test
public void testGetMessage() {
HelloWorld helloWorld=new HelloWorld();
assertEquals("Hello World!!",helloWorld.getMessage());
}

@Test
@Ignore
public void testMyMessage() {
HelloWorld helloWorld=new HelloWorld();
assertEquals("Hello World!!",helloWorld.myMessage());
}
```

Contd…

Contd…

HelloWorldTest

Right-click on your new test class and select *Run-As → JUnit Test*.

```
@After
public void afterMethod() {
System.out.println("After method");
}

@BeforeClass
public static void beforeClass() {
System.out.println("Before all methods");
}

@AfterClass
public static void afterClass() {
System.out.println("After all methods");
}

}
```

Assert Statements in JUnit

- Following are the methods in Assert class :
  - Fail(String)
  - assertTrue(boolean)
  - assertEquals([String message],expected,actual)
  - assertNull([message],object)
  - assertNotNull([message],object)
  - assertSame([String],expected,actual)
  - assertNotSame([String],expected,actual)
  - assertThat(String, T actual, Matcher<T> matcher)

Using @Before and @After

- Test fixtures help in avoiding redundant code when several methods share the same initialization and cleanup code.
- Methods can be annotated with @Before and @After.
  - @Before: This method executes before every test.
  - @After: This method executes after every test.
- Any number of @Before and @After methods can exist.
- They can inherit the methods annotated with @Before and @After.

Using @Before and @After

- Example of @Before:

```
@Before
public void beforeEachTest() {
Calculator cal=new Calculator();
Calculator cal1=new Calculator("5", "2"); }
```

- Example of @After:

```
@After
public void afterEachTest() {
Calculator cal=null;
Calculator cal1=null; }
```

Testing Exceptions

- It is ideal to check that exceptions are thrown correctly by methods.
- Use the expected parameter in @Test annotation to test the exception that should be thrown.
- For example:

```
@Test(expected = ArithmeticException.class)
public void divideByZeroTest() {
calobj.divide(15,0);
}
```

Using @BeforeClass and @AfterClass

- Suppose some initialization has to be done and several tests have to be executed before the cleanup.
- Then methods can be annotated by using the @BeforeClass and @AfterClass.
  - @BeforeClass: It is executed once before the test methods.
  - @AfterClass: It is executed once after all the tests have executed.

Using @BeforeClass and @AfterClass

- Example of @BeforeClass:

```
@BeforeClass
public static void beforeAllTests() {
Connection conn=DriverManager.getConnection(….);}
```

- Example of @AfterClass:

```
@AfterClass
public static void afterAllTests() {
conn.close; }
```

- The methods using this annotation should be public static void

Using @Ignore

- The @Ignore annotation notifies the runner to ignore a test.
- The runner reports that the test was not run.
- Optionally, a message can be included to indicate why the test should be ignored.
- This annotation should be added either in before or after the @Test annotation.

Using @Ignore

- Example of @Ignore for a method:

```
@Ignore ("The network resource is not currently available")
@Test
 public void multiplyTest() {
    ……}
```

- Example of @Ignore for a class:

```
@Ignore
 public class TestCal {
      @Test public void addTest(){ …. }
      @Test public void subtractTest(){…..}
}
```

Unit Testing

- Start with writing tests for methods having the fewest dependencies and then work your way up.
- Ensure that tests are simple, preferably with no decision making.
- Use constant, expected values in the assertions instead of computed values wherever possible.
- Ensure that each unit test is independent of all other tests.
- Clearly document all the tests.
- Test all methods whether public, protected, or private.
- Test the exceptions.

JUnit

- Do not use the constructor of test case to setup a test case, instead use an @Before annotated method.
- Do not assume the order in which tests within a test case should run.
- Place tests and the source code in the same location.

Advanced Testing Concepts

Lesson Objectives

- After completing this lesson, participants will be able to
  - Understand advanced testing concepts
  - Work with test suites
  - Implement parameterized tests

Composing Test into Test Suites

- A Test suite comprises of multiple tests and is a convenient way to group the tests, which are related.
- It also helps in specifying the order for executing the tests.
- JUnit provides the following:
  - org.junit.runners.Suite class : It runs a group of test cases.
  - @RunWith : It specifies runner class to run the annotated class.
  - @Suite.SuiteClasses : It specifies an array of test classes for the Suite.Class to run.
    - The annotated class should be an empty class but may contain initialization and cleanup code.

Composing Test into Test Suites

- Example:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith( Suite.class)
@Suite.SuiteClasses({ TestCalAdd.class, TestCalSubtract.class,
TestCalMultiply.class, TestCalDivide.class })
public class CalSuite {
// the class remains completely empty,
// being used only as a holder for the above annotations
}
```

Test Suite Demo

```
@RunWith(Suite.class)
@Suite.SuiteClasses({ TestPerson1.class,
TestPerson2.class,TestPersonFixture.class})
public class TestPersonSuite {

        @BeforeClass
        public static void setUpBeforeClass() throws Exception {
                System.out.println("Now running the Test Suite");
        }

        @AfterClass
        public static void tearDownAfterClass() throws Exception {
                System.out.println("The Test Suite is completed");
        }

}
```

Parameterized Test class

- Parameterized tests allow you to run the same test with different data.
- To specify parameterized tests:
  - Annotate class with @RunWith(Parameterized.class)
  - Add a public static method that returns a Collection of data.
    - Each element of the collection must be an Array of the various parameters used for the test.
  - Add a public constructor that uses the parameters.

Parameterized Test class

•JUnit allows us to use parameters in a test class.
•This class can contain one test method and this method is executed with the different parameters provided.
•Mark a test class as a parameterized test with the ***@RunWith(Parameterized.class)*** annotation.

•Such a test class must contain a static method annotated with ***@Parameters*** that generates and returns a Collection of Arrays. Each item in this collection is used as the parameters for the test method.

•We should also create a constructor in which we store the values for each test.
•The number of elements in **each** array provided by the method annotated with ***@Parameters*** must correspond to the number of parameters in the constructor of the class.

Reusing Tests

- Example:

```
@RunWith(Parameterized.class)
 public class SomethingTest {
@Parameters
public static Collection<Object[]> data() { …. }
public SomethingTest()
{…..}
@Test
public void testValue()
{…..}
}
```

Parameterized test class

The following code shows an example for a parameterized test.
We will test the multiply() method of the **MyCalculator** class

```
package com.sapient.service;

public class MyCalculator {

public double multiply(double number1,double number2) {
    double result;
    result=number1*number2;
    return result;
}
}
```

Contd…

Parameterized test class

Contd…

If you run this test class, the test method is executed with each defined parameter. In the above example the test method is executed five times

```
@RunWith(Parameterized.class)
public class MyCalculatorParameterized1 {
private double operand1;

public MyCalculatorParameterized1(double operand1) {
    super();
    this.operand1 = operand1;
}

@Parameters
public static Collection<Object[]> data(){
    return Arrays.asList(new Object[][] {
            {25}, {100}, {50}, {10},{5}
        });
}
```

Contd…

Parameterized test class

Contd…

```
@Test
public void testMultiply() {
MyCalculator myCalculator=new MyCalculator();
    assertEquals(operand1*operand1,myCalculator.multiply(operand1,operand1),0.1);
}

}
```

Parameterized test class with 2 parameters

```
@RunWith(Parameterized.class)
public class MyCalculatorParameterized2 {
private double operand1;
private double operand2;

public MyCalculatorParameterized2(double operand1, double operand2)
{
super();
this.operand1 = operand1;
this.operand2 = operand2;
}

@Parameters
public static Collection<Object[]> data(){
return Arrays.asList(new Object[][] {
    {2,3}, {4,5}, {5,6},
    });
}
```

If you run this test class, the test method is executed with each defined parameter. In the this example the test method is executed three times

Contd…

Parameterized test class with 2 parameters

```
@Test
public void testMultiply() {
MyCalculator myCalculator=new MyCalculator();
assertEquals(operand1*operand2,myCalculator.multiply(operand1,
operand2),0.1);
}

}
```

Introduction to Junit 5

Introduction to Junit 5

JUnit Jupiter
This module includes new programming and extension models for writing tests in JUnit 5. New annotations in comparison to JUnit 4 are:

@DisplayName – defines custom display name for a test class or a test method

@BeforeEach – denotes that the annotated method will be executed before each test method (previously *@Before*)

@AfterEach – denotes that the annotated method will be executed after each test method (previously *@After*)

@BeforeAll – denotes that the annotated method will be executed before all test methods in the current class (previously *@BeforeClass*)

@AfterAll – denotes that the annotated method will be executed after all test methods in the current class (previously *@AfterClass*)

@Disable – it is used to disable a test class or method (previously *@Ignore*)

Assertions and Assumptions

JUnit 5 tries to take full advantage of the new features from Java 8, especially lambda expressions

Assertions

Assertions have been moved to ***org.junit.jupiter.api.Assertions*** and have been improved significantly. We can now use lambdas in assertions:

```
Assertions.assertEquals() and Assertions.assertNotEquals()
Assertions.assertArrayEquals()
Assertions.assertIterableEquals()
Assertions.assertLinesMatch()
Assertions.assertNotNull() and Assertions.assertNull()
Assertions.assertNotSame() and Assertions.assertSame()
Assertions.assertTimeout() and Assertions.assertTimeoutPreemptively()
Assertions.assertTrue() and Assertions.assertFalse()
Assertions.assertThrows()
Assertions.fail()
```

Assertions

```
public static void assertEquals(int expected, int actual)

public static void assertEquals(int expected, int actual, String message)

public static void assertEquals(int expected, int actual,
                                Supplier<String< messageSupplier)
```

```
//Test will pass
   Assertions.assertEquals(4, Calculator.add(2, 2));

   //Test will fail
   Assertions.assertEquals(3, Calculator.add(2, 2), "Calculator.add(2, 2) test failed");

   //Test will fail
   Supplier<String> messageSupplier  = ()-> "Calculator.add(2, 2) test failed";
   Assertions.assertEquals(3, Calculator.add(2, 2), messageSupplier);
```

Assertions

```
public static void assertArrayEquals(int[] expected, int[] actual)
public static void assertArrayEquals(int[] expected, int[] actual, String message)
public static void assertArrayEquals(int[] expected, int[] actual,
                                     Supplier<String> messageSupplier)
```

```
@Test
void testCase() {
   //Test will pass
   Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,2,3}, "Array Equal Test");

   //Test will fail because element order is different
   Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,3,2}, "Array Equal Test");

   //Test will fail because number of elements are different
   Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,2,3,4}, "Array Equal Test");
}
```

Assertions

Assertions.assertNotSame() and Assertions.assertSame()

assertNotSame() asserts that **expected and actual DO NOT refer to the same object.** Similarly, assertSame() method asserts that **expected and actual refer to exactly same object.**

```
@Test
void testCase() {
   String originalObject = "Hello World";
   String cloneObject = originalObject;
   String otherObject = "example.com";

   //Test will pass
   Assertions.assertNotSame(originalObject, otherObject);
   //Test will fail
   Assertions.assertNotSame(originalObject, cloneObject);
   //Test will pass
   Assertions.assertSame(originalObject, cloneObject);
   // Test will fail
   Assertions.assertSame(originalObject, otherObject);
}
```

Assertions

Assertions.assertThrows() and Assertions.fail()

It asserts that execution of the supplied Executable throws an exception of the expectedType and returns the exception and fail() method simply fails the test.

```
public static <T extends Throwable> T assertThrows(Class<T> expectedType,
                                                   Executable executable)
```

```
@Test
void testCase() {

   Throwable exception = Assertions.assertThrows(IllegalArgumentException.class,
      () -> {
           throw new IllegalArgumentException("error message");
        });
}
```

```
public class AppTest {
    @Test
    void testCase() {

        Assertions.fail("not found good reason to pass");
        Assertions.fail(AppTest::message);
    }

    private static String message () {
        return "not found good reason to pass";
    }
}
```

Assumptions

Assumptions are used to run tests only if certain conditions are met.

This is typically used for external conditions that are required for the test to run properly, but which are not directly related to whatever is being tested.

You can declare an assumption with *assumeTrue()*, *assumeFalse()*, and *assumingThat().*

Test Suites

JUnit 5 provides two annotations:
@SelectPackages and *@SelectClasses* to create test suites.

Ex.

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.sapient")
public class AllUnitTest {}

@RunWith(JUnitPlatform.class)
@SelectClasses({AssertionTest.class, AssumptionTest.class,
ExceptionTest.class})
public class AllUnitTest {}
```

Introduction To Mockito

Mockito

Mockito is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications.

What is Mocking?

Mocking is a way to test the functionality of a class in isolation.

Mocking does not require a database connection or properties file read or file server read to test a functionality.

Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

What is Mockito?

Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface.

Mock objects are nothing but proxy for actual implementations.

Mockito Annotations

@Mock

The **@Mock** annotation is used to create and inject mocked instances.

We do not create real objects, rather ask mockito to create a mock for the class.

The @Mock annotation is alternative to Mockito.mock(ClassToMock).

Using the @Mock annotation –

- allows shorthand creation of objects required for testing.
- minimizes repetitive mock creation code.
- makes the test class more readable.
- makes the verification error easier to read because field name is used to identify the mock.

@Mock annotation

@Mock Example

We are mocking HashMap class. In real tests, we shall be mocking actual application classes.

```
@Mock
HashMap<String, Integer> mockHashMap;

@Test
public void saveTest(){
   mockHashMap.put("A", 1);

   Mockito.verify(mockHashMap, times(1)).put("A", 1);
   Mockito.verify(mockHashMap, times(0)).get("A");

   assertEquals(0, mockHashMap.size());
}
```

@Spy annotation

@Spy

The **@Spy** annotation is used to **create a real object and spy on that real object**. A spy helps to call all the normal methods of the object while still tracking every interaction, just as we would with a mock.

```
@Spy
HashMap<String, Integer> hashMap;

@Test
public void saveTest(){
   hashMap.put("A", 10);

   Mockito.verify( hashMap, times(1)).put("A", 10);
   Mockito.verify( hashMap, times(0)).get("A");

   assertEquals(1, hashMap.size());
   assertEquals(new Integer(10), (Integer) hashMap.get("A"));
}
```

@Captor annotation

The **@Captor** annotation is used to create an ArgumentCaptor instance which is used to **capture method argument values** for further assertions.

@Captor Example

```
@Mock
HashMap<String, Integer> hashMap;
@Captor
ArgumentCaptor<String> keyCaptor;
@Captor
ArgumentCaptor<Integer> valueCaptor;

@Test
public void saveTest() {
   hashMap.put("A", 10);

   Mockito.verify(hashMap).put( keyCaptor.capture(), valueCaptor.capture());

   assertEquals("A", keyCaptor.getValue());
   assertEquals(new Integer(10), valueCaptor.getValue());
}
```

@Mock and @InjectMocks

@Mock creates a mock whereas @InjectMocks creates an instance of the class and injects the mocks that are created with the @Mock (or @Spy) annotations into this instance.

Note that you must use @RunWith(MockitoJUnitRunner.class) or Mockito.initMocks(this) to initialize these mocks and inject them.

```
@RunWith(MockitoJUnitRunner.class)
public class SomeManagerTest {
@InjectMocks
private SomeManager someManager;
@Mock
private SomeDependency someDependency; // this will be injected into someManager

//tests...
}
```

Thank You!